Introduction to Watson Machine Learning CE

Scaling Up Deep Learning Applications on Summit

Bryant Nelson, Brad Nemanich

February 10, 2020

Distributed Deep Learning on Summit: Overview

- Deep Learning Tools on Summit
- How to Distribute a Model
- Data Handling
- Launching Jobs
- Distributed Deep Learning Best Practices

Distributed Deep Learning on Summit

DEEP LEARNING TOOLS ON SUMMIT

Watson Machine Learning CE

Curated, tested and pre-compiled binary software distribution that enables enterprises to quickly and easily deploy deep learning for their data science and analytics development

Snap ML

Including all of the following frameworks:

Bazel

HDF5

ONNX

Caffe Caffe2

OpenBLAS

TensorFlow
TensorFlow Probability
TensorBoard
TensorFlow-Keras

BVLC Caffe
IBM Enhanced Caffe
Caffe2

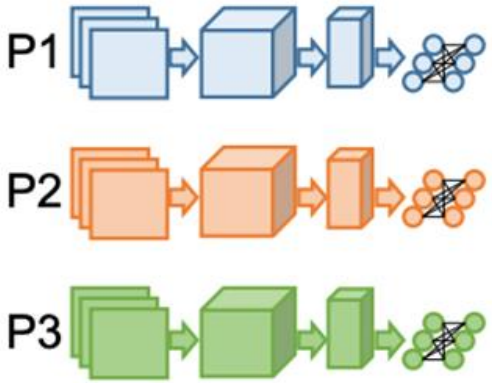
PyTorch

Watson Machine Learning CE: Summit Module

- The WML CE Module can be loaded on Summit with the command
  - module load ibm-wml-ce
- Loading the module will:
  - Activate a conda environment that has all of Watson Machine Learning CE installed
  - Setup conda channels that point to IBM's public conda channel for WML CE, as well as a local conda channel with Summit specific packages
- Users can either use the activated conda environment, or create their own environments from the channels provided by the module

Data Parallel Distributed Deep Learning

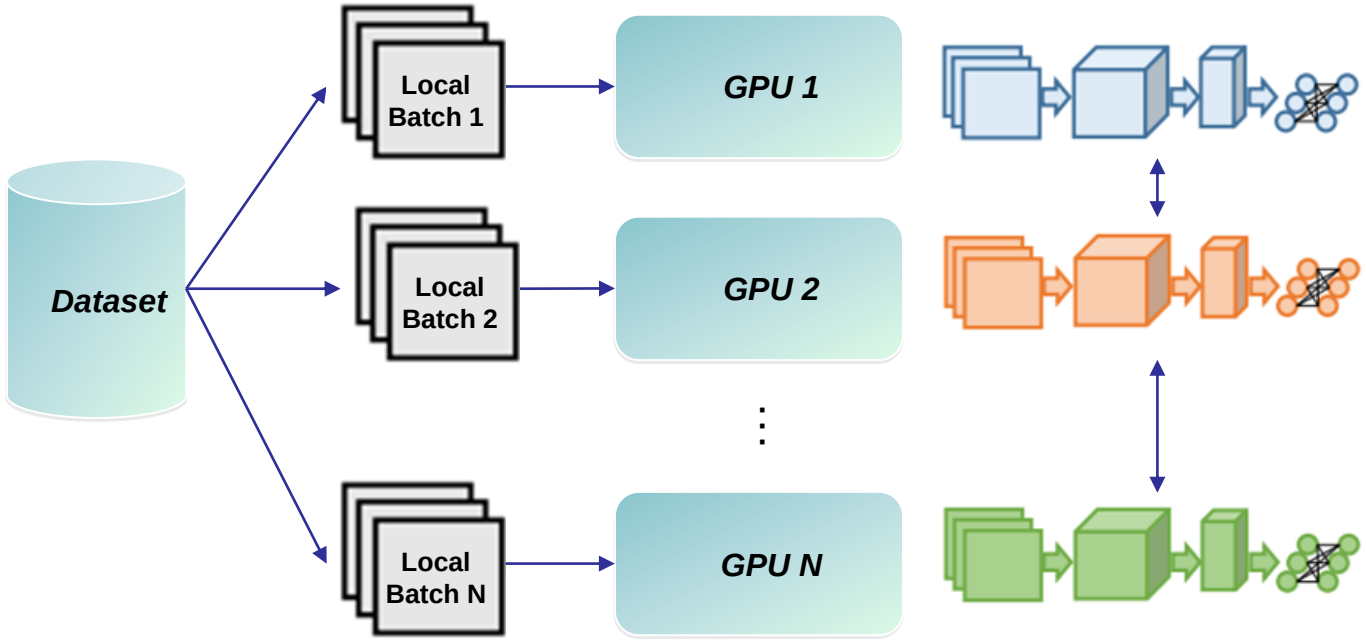
- Synchronous All-to-All Data-Parallel Distributed GPU Deep Learning
- A process is created for each GPU in the cluster
- Each process contains a complete copy of the model
- Mini-batch is spread across all of the processes
  - Each process uses different input data
- After each iteration, all of the processes sync and average together their gradients, and those averages are used to update the local weights.
- Models on each GPU should always be identical.



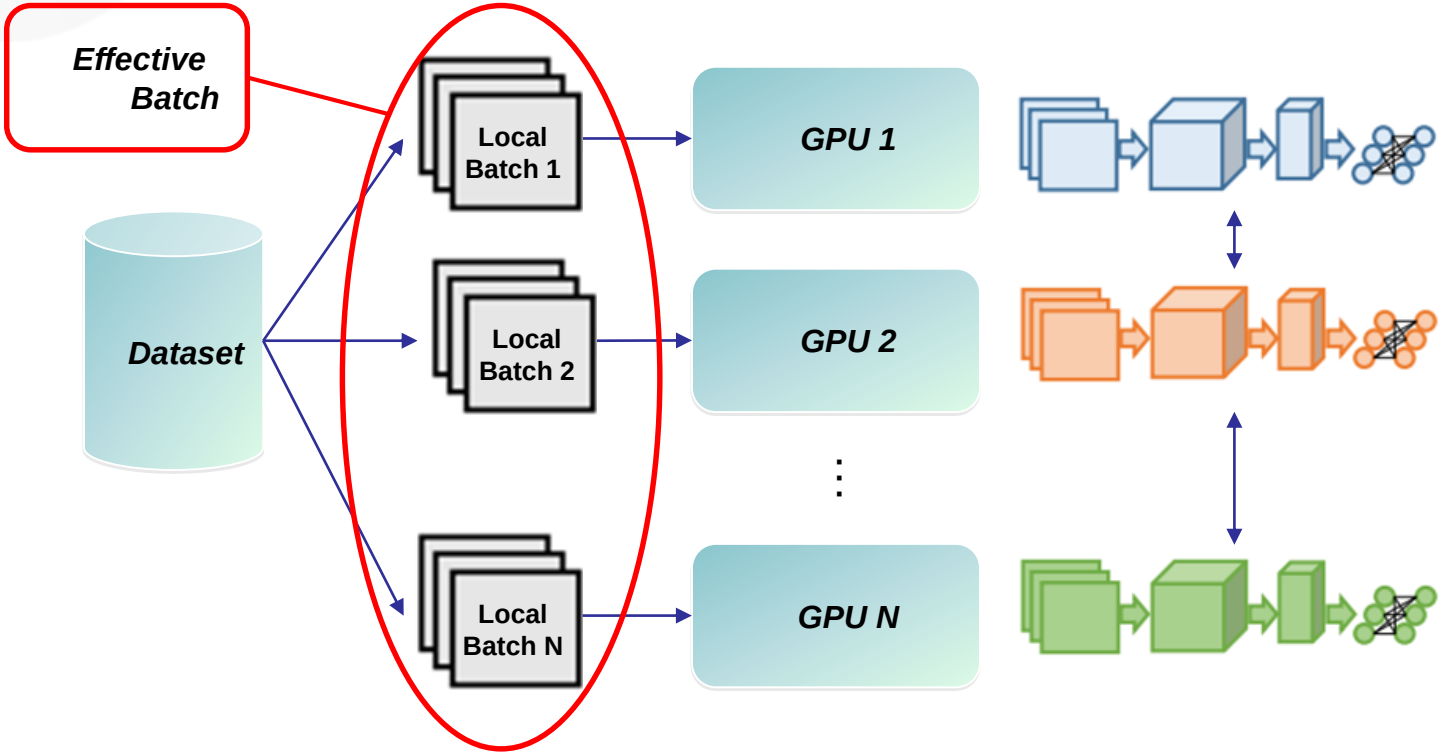
(a) Data Parallelism

[Bergstra et al. 2012]

Data Parallel Distributed Deep Learning



Data Parallel Distributed Deep Learning



Distributed Deep Learning on Summit

HOW TO DISTRIBUTE A MODEL

Horovod

- Horovod is a distributed deep learning training framework for TensorFlow, Keras and PyTorch and can be found in the WML CE module
- Horovod makes it easy to enable distributed deep learning within an existing script, while providing very good scaling across a cluster
- Horovod can use different communication libraries to talk between "learners", including MPI, NCCL, IBM DDL and GLOO

Enable Distributed Deep Learning with Horovod

- In order to use Horovod to perform a distributed deep learning job, the following steps must be taken
  - Initialize Horovod
  - Select the GPU to use based on the local rank of the job
  - Scale the learning rate
  - Add the Horovod optimizer
  - Broadcast global variables
  - Modify scripts to only perform checkpointing (and maybe logging) on rank 0

Enable Distributed Deep Learning with Horovod: Initialize Horovod

- To initialize Horovod in TensorFlow, add the following to your script:

```
import horovod.tensorflow as hvd

hvd.init()
```

- To initialize Horovod in PyTorch, add the following to your script:

```
import horovod.torch as hvd

hvd.init()
```

Enable Distributed Deep Learning with Horovod: Select a GPU to Run On

- Select the GPU to use in TensorFlow:

```
config = tf.ConfigProto()
config.gpu_options.visible_device_list = str(hvd.local_rank())
```

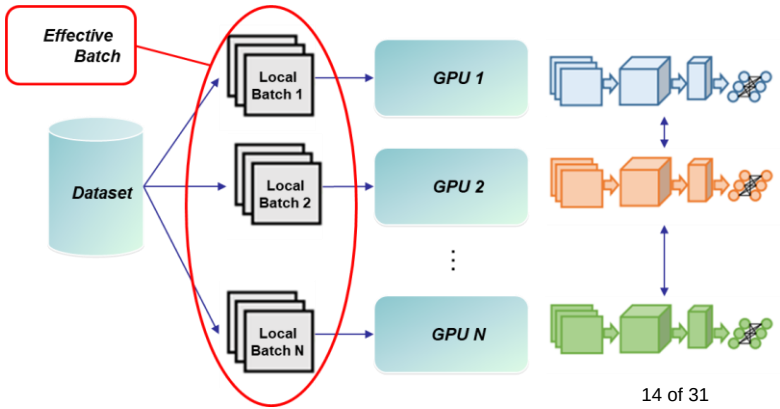
- Select the GPU to use in PyTorch:

```
torch.cuda.set_device(hvd.local_rank())
```

Enable Distributed Deep Learning with Horovod: Scale the Learning Rate

- In data parallel distributed deep learning, the effective batch size increases with the number of "learners". Some hyper-parameters will need to be modified to account for this larger batch size.
- Normally, we scale the learning rate by the total number of “learners” to offset the effect of the larger global batch size.
  - hvd.size() can be used to get the total number of learners:

```
learn_rate*=hvd.size()
```



Enable Distributed Deep Learning with Horovod: Add the Horovod Optimizer

- The Horovod optimizer enables the "learners" to sync-up.
  - In TensorFlow:

```
opt = hvd.DistributedOptimizer(opt)
```

  - In PyTorch

```
optimizer = hvd.DistributedOptimizer(optimizer, named_parameters=model.named_parameters())
```

Enable Distributed Deep Learning with Horovod: Broadcast Global Variables

- It is important that each "learner's" variables are initialized with the same values.
- To accomplish this, learner 0 broadcasts the values of all of its trainable variables:
  - In TensorFlow:

```
hooks = [hvd.BroadcastGlobalVariablesHook(0)]
```

  - In PyTorch:

```
hvd.broadcast_parameters(model.state_dict(), root_rank=0)
```

Enable Distributed Deep Learning with Horovod: Broadcast Global Variables

- It is important that each "learner's" variables are initialized with the same values.
- To accomplish this, learner 0 broadcasts the values of all of its trainable variables:
  - In TensorFlow:

```
hooks = [hvd.BroadcastGlobalVariablesHook(0)]
```

  - In PyTorch:

```
hvd.broadcast_parameters(model.state_dict(), root_rank=0)
```

Enable Distributed Deep Learning with Horovod: Restrict Checkpointing to Rank 0

- Since there are multiple processes per node, it is important to make sure that the are no file access errors from multiple processes trying to write checkpoints at the same time.

```
checkpoint_dir = '/mnt/bb/$USER/train_logs' if hvd.rank() == 0 else None
```

- It is also possible to limit the verbosity on all but rank 0 to clean up the output.

```
verbose = hvd.rank() == 0
```

Distributed Deep Learning on Summit

DATA HANDLING

Data Handling

- To gain any benefit from distributed training, each learner needs to train on different data. Otherwise each learner is doing the same work.
- There are two common approaches to handling data for distribution:
  - Data Splitting
  - Data Sampling
- If the model is already randomly shuffling/sampling its data set, it may be that no data handling changes are needed to distribute training.

Data Handling: Data Splitting

- One way to ensure that each learner trains on different data is to split the training data.
  - This can be done offline, before training, and the correct split loaded at runtime:

```
x_train = np.load('train_data_' + str(hvd.rank()) + '.npy')
y_train = np.load('train_labels_' + str(hvd.rank()) + '.npy')
```

  - Or the data can be split at runtime:

```
x_train = np.array_split(x_train, hvd.size())[hvd.rank()]
y_train = np.array_split(y_train, hvd.size())[hvd.rank()]
```

- If your model makes use of batch normalization it is important that each data split has the same distribution as the full dataset.

Data Handling: Data Sampling

- Another way to ensure that each learner trains on different data is to randomly sample each local batch from the training data at runtime.
  - This works well if the batch sizes are significantly smaller than the overall size of the data set.
  - To ensure that each learner is seeing different data, unique seeds based on rank should be used.

```
np.random.seed(SEED + hvd.rank())
…
sample = np.random.choice(x_train.shape[0], size=batch_size)
x_train = x_train[sample]
y_train = y_train[sample]
```

Distributed Deep Learning on Summit

LAUNCHING JOBS

Launching Jobs: ddlrun

- Watson Machine Learning CE provides ddlrun:
  - interfaces with lsf to get cluster information
  - launches job across all nodes in reservation
  - activates the active conda environment on all compute nodes

```
[brynelso@login2.summit ~]$ bsub -W 30 -nnodes 2 -P ven201 -Is bash
Job <891318> is submitted to default queue <batch>.
<<Waiting for dispatch ...>>
<<Starting on batch1>>
bash-4.2$ module load ibm-wml-ce
(ibm-wml-ce-1.6.2-2) bash-4.2$ ddlrun python pytorch_synthetic.py
+ /sw/summit/xalt/1.1.4/bin/mpirun -x LSB_MCPU_HOSTS -x PATH -x PYTHONPATH -x LD_LIBRARY_PATH -x LSB_JOBID -
disable_gdr -gpu -x NCCL_TREE_THRESHOLD=0 -x NCCL_LL_THRESHOLD=0 -mca plm_rsh_num_concurrent 2 --rankfile
/tmp/DDLRUN/DDLRUN.hNKBnciW3ROH/RANKFILE -n 12 -x DDL_HOST_PORT=2200 -x
"DDL_HOST_LIST=h20n18:0,2,4,6,8,10;h30n06:1,3,5,7,9,11" -x "DDL_OPTIONS=-mode p:6x2x1x1 " bash -c 'source
/sw/summit/ibm-wml-ce/anaconda-base/etc/profile.d/conda.sh && conda activate /sw/summit/ibm-wml-ce/anaconda-
base/envs/ibm-wml-ce-1.6.2-2 > /dev/null 2>&1 && python pytorch_synthetic.py'
...
```

Distributed Deep Learning on Summit

BEST PRACTICES

Best Practices: How to Stage to NVME

- This BSUB script:
  - stages data to each compute nodes local NVME drive
  - runs a training job across all nodes

```
#!/bin/bash
#BSUB -W 0:20
#BSUB -P <project>
#BSUB -q batch
#BSUB -J ddl_test
#BSUB -nnodes 18
#BSUB -alloc_flags NVME
module load ibm-wml-ce

ddlrun --accelerators 1 bash -c 'cp -Ra /gpfs/alpine/proj-shared/<project>/imagenetTF /mnt/bb/$USER'

ddlrun python $CONDA_PREFIX/horovod/examples/pytorch_imagenet_resnet50.py \
            --train-dir /mnt/bb/$USER/imagenetTF/train \
            --val-dir /mnt/bb/$USER/imagenetTF/val
```

Best Practices: How to Stage to NVME

- -alloc_flags NVME gives access to the NVME drives
- The NVME filesystem can be accessed at /mnt/bb/$USER

```
#!/bin/bash
#BSUB -W 0:20
#BSUB -P <project>
#BSUB -q batch
#BSUB -J ddl_test
#BSUB -nnodes 18
#BSUB -alloc_flags NVME
module load ibm-wml-ce

ddlrun --accelerators 1 bash -c 'cp -Ra /gpfs/alpine/proj-shared/<project>/imagenetTF /mnt/bb/$USER

ddlrun python $CONDA_PREFIX/horovod/examples/pytorch_imagenet_resnet50.py \
            --train-dir /mnt/bb/$USER/imagenetTF/train \
            --val-dir /mnt/bb/$USER/imagenetTF/val
```

Best Practices: Synchronized Scheduling and Early Stopping

- By default only the gradients, and by extension the weights, are synchronized between processes.
- To ensure things like early stopping and velocity based learning rate schedules are the same across nodes some additional synchronizing is needed.
  - In tf.keras this can be accomplished with the MetricAverageCallback

```
callbacks=[hvd.callbacks.BroadcastGlobalVariablesCallback(0),
           hvd.callbacks.MetricAverageCallback(),
           ModelCheckpoint(checkpoint_file, monitor='val_loss', save_best_only=True),
           ReduceLROnPlateau(monitor='val_loss', factor=0.5, patience=2, min_lr=lrate*0.0001),
           EarlyStopping(patience=5, min_delta=0.0001)])
```

Best Practices: Distribute On-the-Fly Validation

- Most frameworks support some method of on-the-fly validation.
- If the validation is not also distributed, it will quickly bottleneck the training.
- In tf.keras this can be accomplished by splitting or sampling the validation data in the same manner as the training data, and using the MetricAverageCallback to average the validation loss.
- In other frameworks it may be necessary to explicitly average the validation loss from each node.

Thank You

Bryant Nelson
Brad Nemanich
—
bryant.nelson@ibm.com
brad.nemanich@ibm.com
http://www.ibm.com

Documentation and Resources

- OLCF IBM Watson Machine Learning CE documentation:
  - https://docs.olcf.ornl.gov/software/analytics/ibm-wml-ce.html

- IBM WML CE Public Conda Channel with x86 Support:
  - https://public.dhe.ibm.com/ibmdl/export/pub/software/server/ibm-ai/conda/
  - Usage Documentation:
    - https://www.ibm.com/support/knowledgecenter/SS5SF7_1.6.2/navigation/wmlce_install.htm
- IBM CML CE Early Access Channel:
  - https://public.dhe.ibm.com/ibmdl/export/pub/software/server/ibm-ai/conda-early-access/
  - Usage Documentation:
    - https://developer.ibm.com/linuxonpower/2020/02/10/using-the-watson-machine-learning-community-edition-auxiliary-channels/